CS 320: Concepts of Programming Languages

Wayne Snyder
Computer Science Department
Boston University

Lecture 13: Implementing a Language: Syntaxc

- Context-Free Grammars
- Derivations and Derivation Trees
- Parse Trees and Parsing
- Abstract Syntax Trees (ASTs)
- Top-Down Parsing

Syntax of Programming Languages: Languages

In computer science, a **language** is any set of strings from a fixed alphabet. In general, we'll assume that the alphabet is the set of ASCII characters.

We can specify languages informally in English:

Informal Description	Set of Strings
Names of weekdays	{ "Monday", "Tuesday", "Wednesday", "Thursday", "Friday" }
Bit strings of length 3	{ "000", "001", "010", "011", "100", "101", "110", "111" }
Strings consisting of a's and b's with an equal number of each:	{ "", "ab", "ba", "aabb", "abab", "abba", "bbaa", "aaabbb", ... }
All strings of matching parentheses:	{ "()", "(())", "()()", "()()()", "(())()", "()(())", ... }

Q: Is there a better way to specify a language than either describing them in English or simply listing all the strings? Not really practical when the language is infinite!

Context-Free Grammars

A Context-Free Grammar (also called Backus-Naur Form, BNF) is the primary way that the syntax of a programming language is specified. It consists of the following components:

T – A set of **Terminal Symbols**, which for us will simply be the alphabet of ASCII characters. Notation:

- letters: a b c q 'A' 'T'
- digits: 1 5 9
- arithmetic operators: * + - /
- punctuation: ( ) , ; } {

We will tend to leave off quote marks ' and " when no confusion would result.

N – A set of **Non-Terminal Symbols**, representing sets of strings of terminal symbols (i.e., languages). Notation:

- capital letters: S E F T
- descriptions in angle brackets: <expressions> <ops> <statements>

One particular Non-Terminal (often S) is called the **Start Symbol**.

Context-Free Grammars

A Context-Free Grammar (also called Backus-Naur Form, BNF) is the primary way that the syntax of a programming language is specified. It consists of the following components:

T – A set of **Terminal Symbols**, which for us will simply be the alphabet of ASCII characters.

N – A set of **Non-Terminal Symbols**, representing sets of strings of terminal symbols (i.e., languages). S ∈ N is called the **Start Symbol**.

P – A set of **Production Rules**, of the form:

Non-Terminal -> String of terminals and/or non-terminal symbols (possibly empty)

Examples:

S -> a

<expr> -> <expr> + <number>

S -> a S a

A -> ε (ε is the empty string "")

Context-Free Languages

For any grammar G, the language **L(G)** is called a **Context-Free Language**, and is generated by G by deriving a string of terminal symbols from the start symbol by using the production rules to **rewrite** a string of symbols, starting with the start symbol.

Example: P = 0: S -> a S a T = { a, b } N = { S }
1: S -> b S b
2: S -> a
3: S -> b
4: S -> ε

Here is a **derivation** of the string ababa, with each rewrite step labelled by the number of the production rule used, and where the symbol rewritten is underlined:

$$\underline{S} \Rightarrow_0 a\underline{S}a \Rightarrow_1 ab\underline{S}ba \Rightarrow_2 ababa$$

Here is a **derivation** of the string aaaaaa:

$$\underline{S} \Rightarrow_0 a\underline{S}a \Rightarrow_0 aa\underline{S}aa \Rightarrow_0 aaa\underline{S}aaa \Rightarrow_4 aaaaaa$$

Context-Free Languages

Example: The language of matching parentheses:

P = 0: S -> (S) T = { (,) } N = { S }
1: S -> ( )
2: S -> S S

Derivation of ((())):

$$\underline{S} \;\Rightarrow_0\; (\underline{S}) \;\Rightarrow_0\; ((\underline{S})) \;\Rightarrow_1\; ((()))$$

Derivation of ((()))():

$$\underline{S} \;\Rightarrow_2\; \underline{S}S \;\Rightarrow_0\; (\underline{S})S \;\Rightarrow_0\; ((\underline{S}))S \;\Rightarrow_1\; ((()))\underline{S} \;\Rightarrow_1\; ((()))()$$

Context-Free Languages

Example: The language of arithmetic expressions over +, ∗, and single digits:

P = 0: S -> E
1: E -> E + E
2: E -> E ∗ E
3: E -> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 (shorthand for E -> 0, E -> 1, etc.)

T = { +, ∗, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 }
N = { S, E }

Derivation of 8 ∗ 2 + 7:

$$\underline{S} \Rightarrow_0 \underline{E} \Rightarrow_2 E * \underline{E} \Rightarrow_1 \underline{E} * E + E$$

$$\Rightarrow_3 8 * \underline{E} + E$$

$$\Rightarrow_3 8 * 2 + \underline{E}$$

$$\Rightarrow_3 8 * 2 + 7$$

Context-Free Languages

0: S -> E
1: E -> E + E
2: E -> E * E
3: E -> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Note that there may be more than one derivation for a given string:

Derivations of 8 * 2 + 7:

$$\underline{S} =>_0 \underline{E} =>_2 E * \underline{E} =>_1 \underline{E} * E + E =>_3 8 * \underline{E} + E =>_3 8 * 2 + \underline{E} =>_3 8 * 2 + 7$$

$$\underline{S} =>_0 \underline{E} =>_1 \underline{E} + E =>_2 \underline{E} * E + E =>_3 8 * \underline{E} + E =>_3 8 * 2 + \underline{E} =>_3 8 * 2 + 7$$

$$\underline{S} =>_0 \underline{E} =>_2 E * \underline{E} =>_3 E * E + \underline{E} =>_1 E * \underline{E} + 7 =>_3 \underline{E} * 2 + 7 =>_3 8 * 2 + 7$$

Two simple strategies are:

- **Left-most Derivation**: The left-most non-terminal is rewritten at each step.
- **Right-most Derivation**: The right-most non-terminal is rewritten at each step.

Which is which in the above examples?

Context-Free Languages

0: S -> E
1: E -> E + E
2: E -> E * E
3: E -> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Note that there may be more than one derivation for a given string:

Derivations of 8 * 2 + 7:

$$\underline{S} \Rightarrow_0 \underline{E} \Rightarrow_2 E * \underline{E} \Rightarrow_1 \underline{E} * E + E \Rightarrow_3 8 * \underline{E} + E \Rightarrow_3 8 * 2 + \underline{E} \Rightarrow_3 8 * 2 + 7$$

Left-Most:

$$\underline{S} \Rightarrow_0 \underline{E} \Rightarrow_1 \underline{E} + E \Rightarrow_2 \underline{E} * E + E \Rightarrow_3 8 * \underline{E} + E \Rightarrow_3 8 * 2 + \underline{E} \Rightarrow_3 8 * 2 + 7$$

Right-Most:

$$\underline{S} \Rightarrow_0 \underline{E} \Rightarrow_2 E * \underline{E} \Rightarrow_1 E * E + \underline{E} \Rightarrow_3 E * \underline{E} + 7 \Rightarrow_3 \underline{E} * 2 + 7 \Rightarrow_3 8 * 2 + 7$$

Two simple strategies are:

- **Left-most Derivation**: The left-most non-terminal is rewritten at each step.
- **Right-most Derivation**: The right-most non-terminal is rewritten at each step.

Which is which in the above examples?

Context-Free Languages

0: S -> E
1: E -> E + E
2: E -> E * E
3: E -> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

In general we will focus on left-most derivations. Note that a string may have more than one left-most derivation:

Left-most derivations of 8 * 2 + 7:

$$\underline{S} \Rightarrow_0 \underline{E} \Rightarrow_1 \underline{E} + E \Rightarrow_2 \underline{E} * E + E \Rightarrow_3 8 * \underline{E} + E \Rightarrow_3 8 * 2 + \underline{E} \Rightarrow_3 8 * 2 + 7$$

$$\underline{S} \Rightarrow_0 \underline{E} \Rightarrow_2 \underline{E} * E \Rightarrow_3 8 * \underline{E} \Rightarrow_1 8 * \underline{E} + E \Rightarrow_3 8 * 2 + \underline{E} \Rightarrow_3 8 * 2 + 7$$

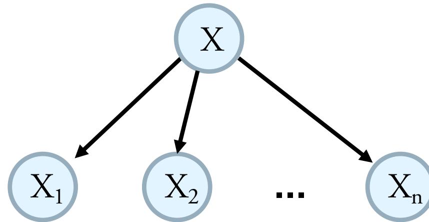
The best way to think about derivations is to visualize them using derivation trees....

Context-Free Languages

0: S -> E
1: E -> E + E
2: E -> E * E
3: E -> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

The set of derivation trees T(G) for a grammar G can be defined inductively:

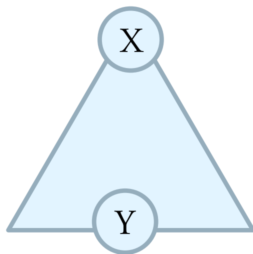
Base Case: For every production rule $X \rightarrow X_1 X_2 \ldots X_n$ in G, the following tree is in T(G):

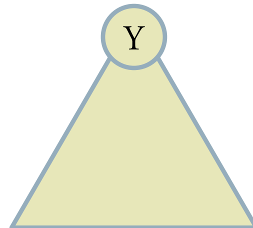


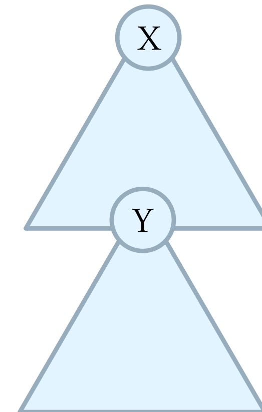
$X \in N$

$\{ X_1, X_2, \ldots, X_n \} \subseteq N \cup T$

Inductive Case: If T(G) contains the two derivation trees on left with non-terminals X and Y at their roots, where Y occurs as a leaf node as shown, then T(G) also contains the tree on the right:



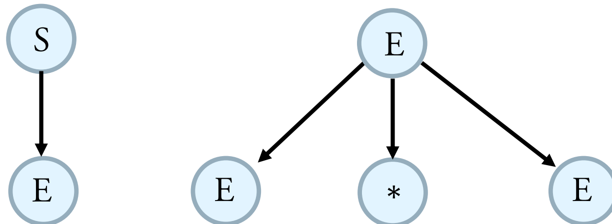


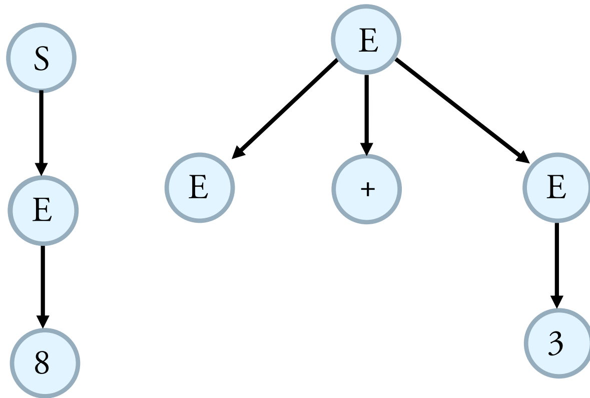


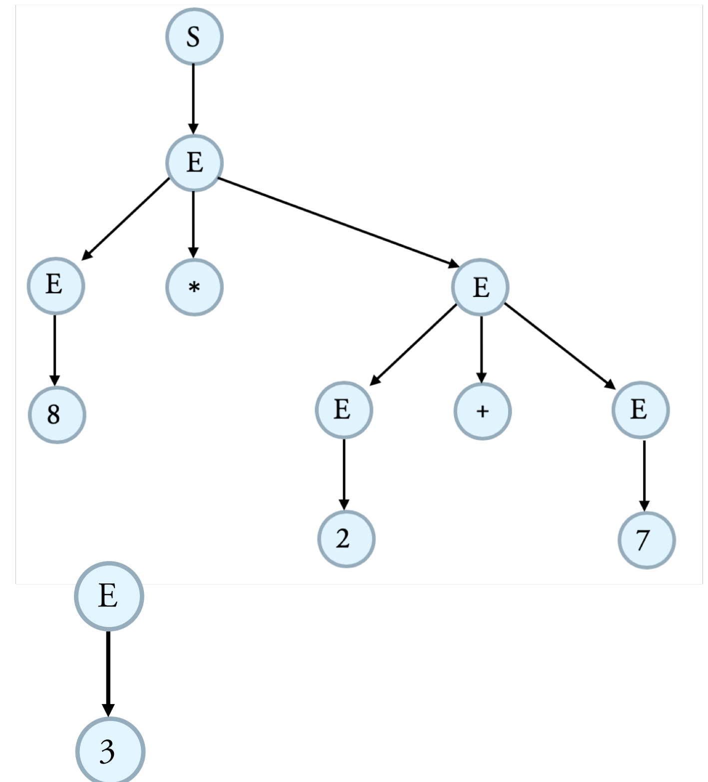
Context-Free Languages

0: S -> E
1: E -> E + E
2: E -> E * E
3: E -> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Examples: The following are examples of derivation trees for our arithmetic grammar.



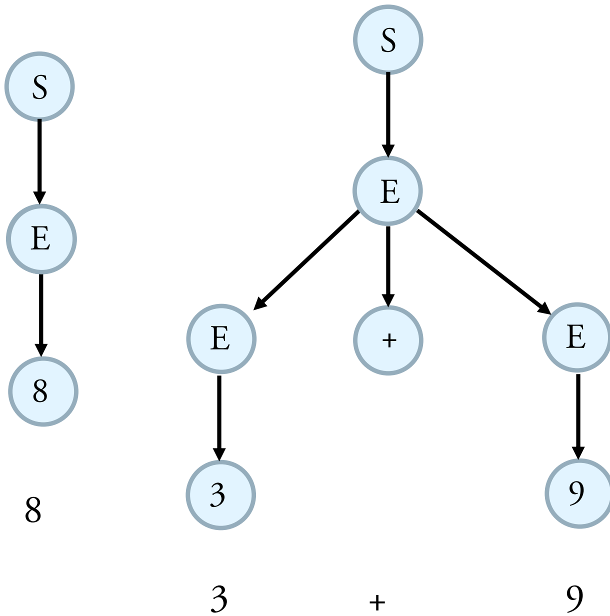


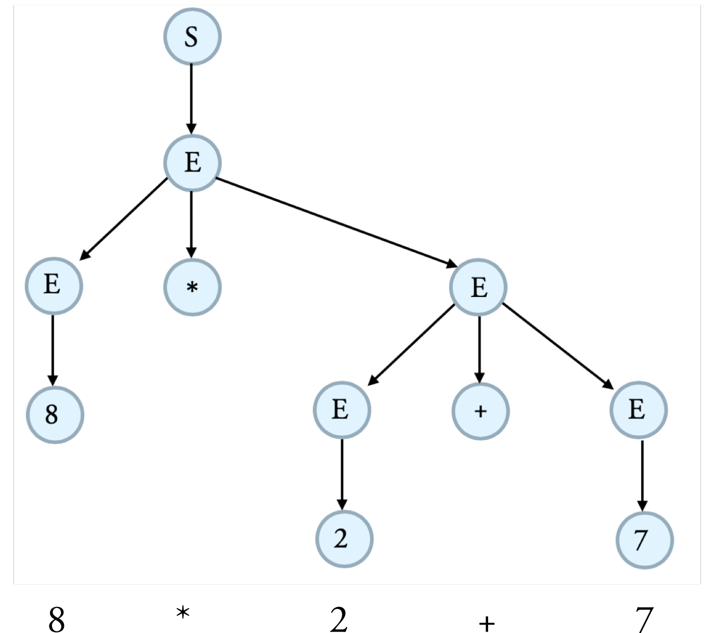


Context-Free Languages

0: S -> E
1: E -> E + E
2: E -> E * E
3: E -> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

A derivation tree is called a parse tree for a string w of terminals if it has S at the root, and the leaves (left to right) "spell" the string w. Examples:



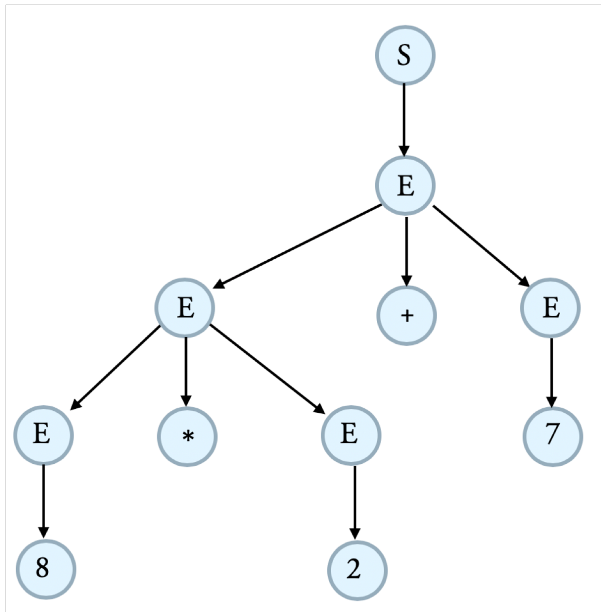


Digression: Parse Trees vs. Abstract Syntax Trees (ASTs)

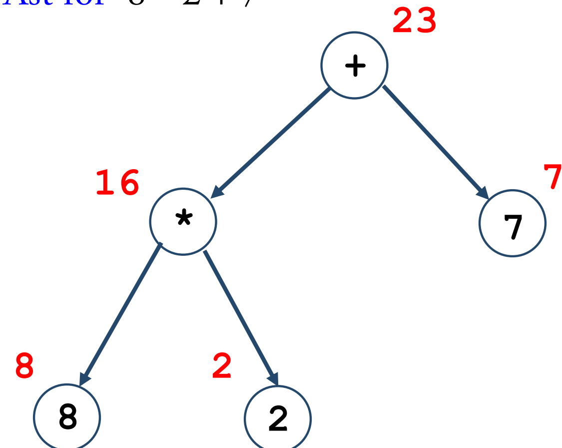
Parse trees are sometimes called "Concrete Syntax Trees" to distinguish them from Abstract Syntax Trees (ASTs).

Parse trees represent the syntax of programming language expressions;
ASTs represent the semantics (the computational meaning) of expressions.

Parse tree for 8 * 2 + 7



Ast for 8 * 2 + 7



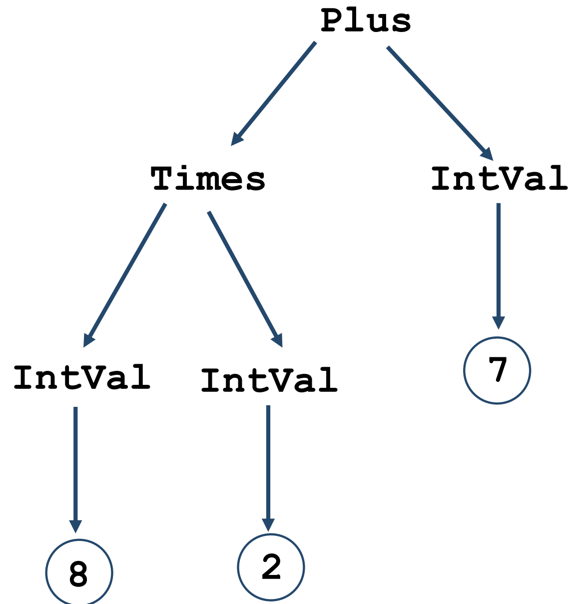
Digression: Parse Trees vs. Abstract Syntax Trees (ASTs)

As you have seen, Haskell has an elegant and simple way of representing ASTs, since they just correspond to user-created data types:

```
(Plus (Times (IntVal 8) (IntVal 2)) (IntVal 7) )

data Ast =
   IntVal Integer
 | Plus Ast Ast
 | Times Ast Ast
```

Parsing is the process of discovering the structure of the parse tree for a string and converting it into an Ast, which can then be evaluated.

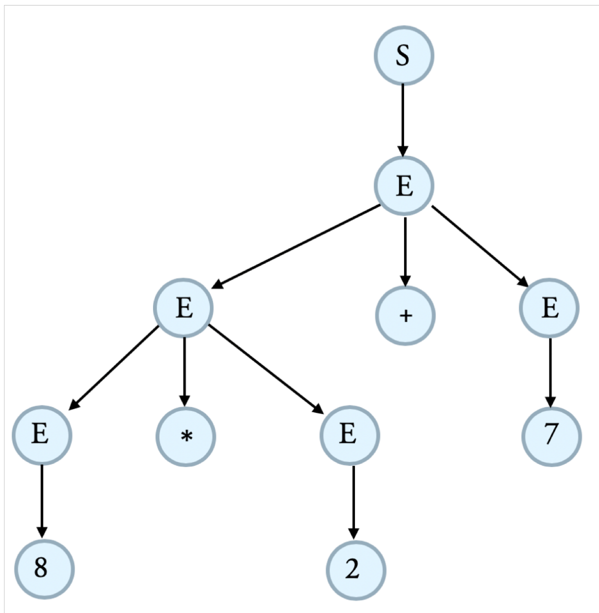


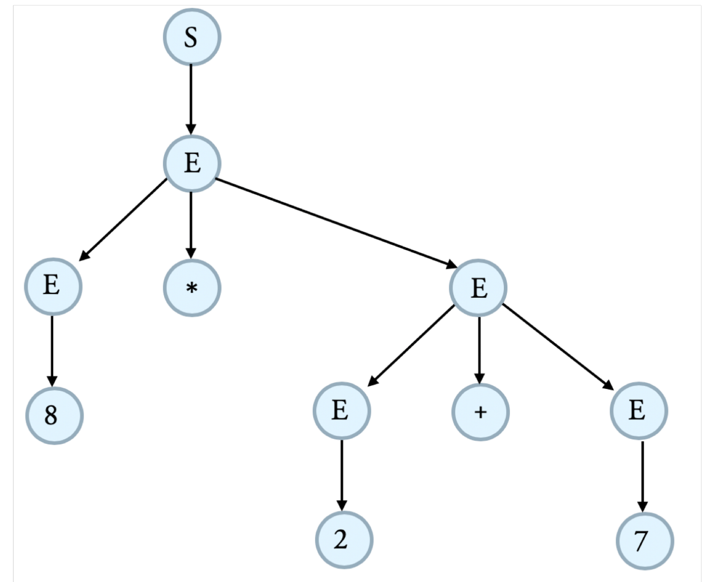
(End of Digression)

Context-Free Languages

0: S -> E
1: E -> E + E
2: E -> E * E
3: E -> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Note that for this grammar, there can be more than one parse tree for the same string:

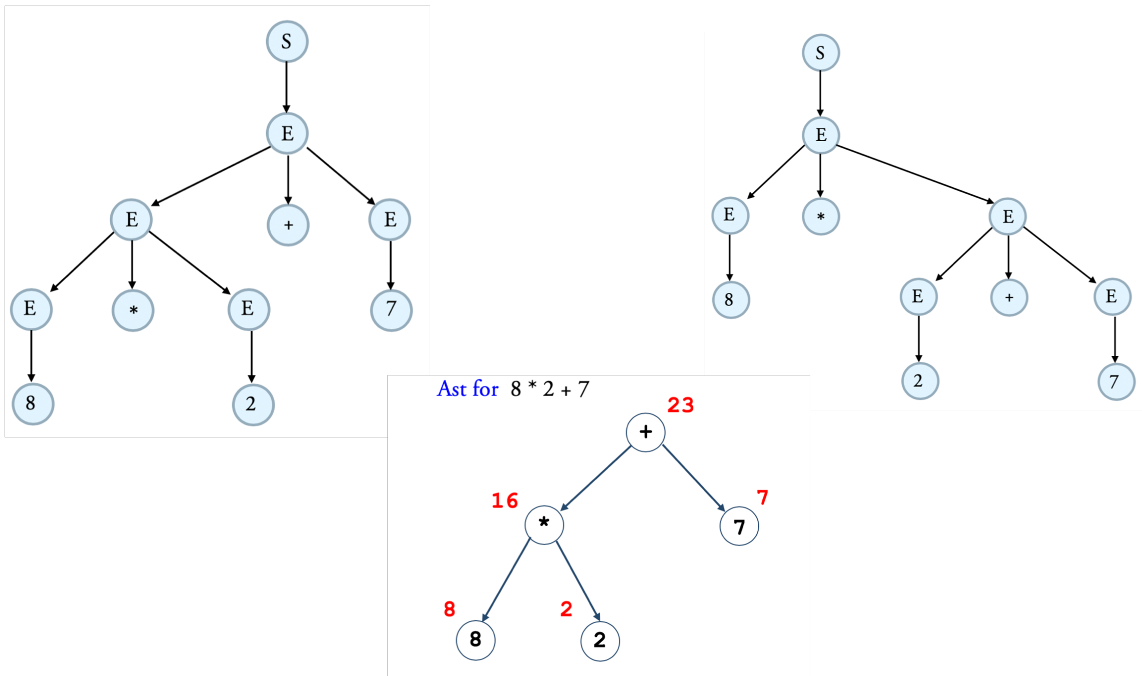




A grammar is said to be ambiguous if there is a string which has two distinct parse trees.

Context-Free Languages

Ambiguity is a serious problem, since it affects the way we interpret an expression, and makes it difficult to translate from concrete syntax to abstract syntax:



Clearly, the parse tree on the left is closer to the actual meaning of the expression.

Context-Free Languages

In order to solve this problem, we have to "hack" a different grammar which avoids these ambiguities. We will look only at the issue of precedence and associativity of operators, although there are other ways that grammars can be ambiguous.

There is no problem with evaluation order in a fully-parenthesized expression:

$$(2 - ((8 * 2) + 7))$$

But in order to enhance readability, in programming languages (and math more generally) we may eliminate parentheses, and precedence and associativity rules tell us in which order to perform the operations:

Precedence class

<table>
<tr><th>Precedence</th><th>Operator</th><th>Description</th><th>Associativity</th></tr>
<tr><td>9
highest</td><td><code>.</code></td><td>Function composition</td><td>Right</td></tr>
<tr><td>8</td><td><code>^,^^,**</code></td><td>Power</td><td>Right</td></tr>
<tr><td>7</td><td><code>*,/,`quot`,`rem`,`div`,`mod`</code></td><td></td><td>Left</td></tr>
<tr><td>6</td><td><code>+,-</code></td><td></td><td>Left</td></tr>
<tr><td>5</td><td><code>:</code></td><td>Append to list</td><td>Right</td></tr>
<tr><td>4</td><td><code>==,/=,<,<=,>=,></code></td><td>Compare-operators</td><td></td></tr>
<tr><td>3</td><td><code>&&</code></td><td>Logical AND</td><td>Right</td></tr>
<tr><td>2</td><td><code>||</code></td><td>Logical OR</td><td>Right</td></tr>
<tr><td>1</td><td><code>>>,>>=</code></td><td></td><td>Left</td></tr>
<tr><td>1</td><td><code>=<<</code></td><td></td><td>Right</td></tr>
<tr><td>0</td><td><code>$,$!,`seq`</code></td><td></td><td>Right</td></tr>
</table>

Associativity of the precedence class

Context-Free Languages

Precedence class

Precedence	Operator	Description	Associativity
9 highest	`.`	Function composition	Right
8	`^,^^,**`	Power	Right
7	``*,/,`quot`,`rem`,`div`,`mod` ``		Left
6	`+,-`		Left
5	`:`	Append to list	Right
4	`==,/=,<,<=,>=,>`	Compare-operators	
3	`&&`	Logical AND	Right
2	`||`	Logical OR	Right
1	`>>,>>=`		Left
1	`=<<`		Right
0	``$,$!,`seq` ``		Right

Associativity of the precedence class

By using ':info' command, you can get the precedence levels of operators.

```
*Main> :info +
class Num a where
  (+) :: a -> a -> a
  ...
   -- Defined in 'GHC.Num'
infixl 6 +
*Main>
*Main> :info ++
(++) :: [a] -> [a] -> [a]   -- Defined in 'GHC.Base'
infixr 5 ++
*Main>
*Main> :info ==
class Eq a where
  (==) :: a -> a -> Bool
  ...
   -- Defined in 'GHC.Classes'
infix 4 ==
```

If you declare your own operator, you can define its precedence and associativity using the functions `infixl`, `infixr`, and `infix`; if you do not specify, it is considered to be highest precedence (9) and left-associative.

Context-Free Languages

Example:

2 - 8 ^ 4 ^ 3 + 7 = 2 - (8 ^ 4 ^ 3) + 7

= 2 - (8 ^ (4 ^ 3)) + 7

= (2 - (8 ^ (4 ^ 3))) + 7

= ((2 - (8 ^ (4 ^ 3))) + 7)

Precedence	Operator	Description	Associativity
9 highest	`.`	Function composition	Right
8	`^,^^,**`	Power	Right
7	``*,/,`quot`,`rem`,`div`,`mod` ``		Left
6	`+,-`		Left
5	`:`	Append to list	Right
4	`==,/=,<,<=,>=,>`	Compare-operators	
3	`&&`	Logical AND	Right
2	`||`	Logical OR	Right
1	`>>,>>=`		Left
1	`=<<`		Right
0	``$,$!,`seq` ``		Right

Precedence class

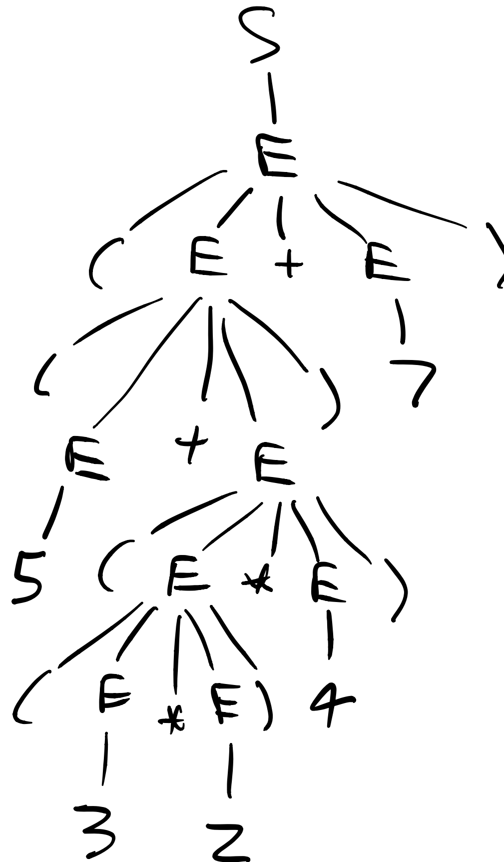
Associativity of the precedence class

Context-Free Languages

If we full parenthesize our expressions, then it is easy to write a non-ambiguous grammar:

0: S -> E
1: E -> (E + E)
2: E -> (E * E)
3: E -> 0 | 1 | 2 | ... | 9

((5 + ((3 * 2) * 4)) + 7)



Context-Free Languages

But to solve the more general problem, we have to use a couple of tricks to modify the grammar:

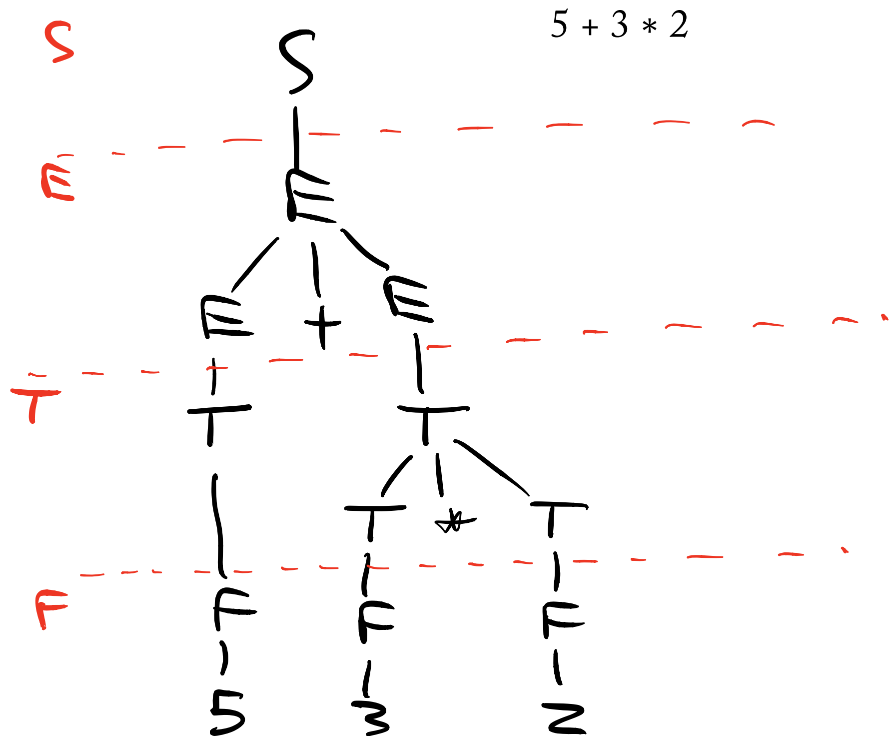
(1) To enforce precedence, we have to "stratify" (or "factor") a grammar into separate strata, using separate non-terminals, so that the parse tree always has lower precedence operators at top of tree and higher at the bottom:

0: S -> E

1: E -> E + E
2: E -> T

3: T -> T * T
4: T -> F

5: F -> 0 | 1 | ... | 9 |



Context-Free Languages

But to solve the more general problem, we have to use a couple of tricks to modify the grammar:

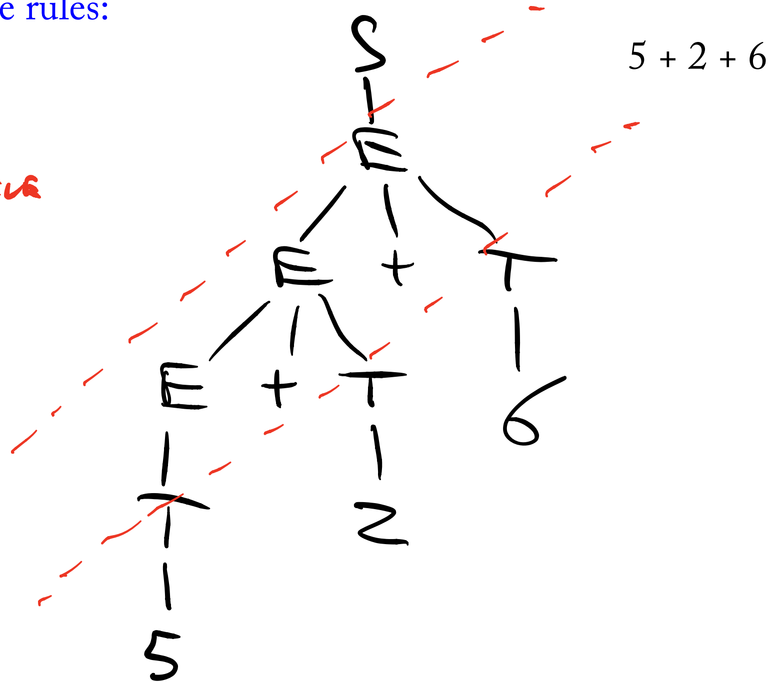
(2) To enforce associativity, we have arrange the recursion in the rules so that the left non-terminal is parsed first; For left-associativity we use left-recursive rules, and for right-recursive we use right-recursive rules:

0: S -> E

1: E -> E + T
2: E -> T

3: T -> 0 | 1 | ... | 9 |

LEFT-RECURSIVE RULE



Context-Free Languages

Putting these two techniques together, we arrive at a non-ambiguous grammar for operators such as we have in Haskell:

5 - 2 * 8 / 3 + ~ 6 ^ 2 ^ 4

0: S -> E

1: E -> E + T
2: E -> E - T
3: E -> T

4: T -> T * F
5: T -> T / F
6: T -> F

7: F -> P ^ F
8: F -> P

9: P -> ~ P
10: P -> M

11: M -> 0 | 1 | ... | 9 |

Context-Free Languages

FINALLY, if we want to have parentheses, which force us to start all over again with the lowest-precedence operators, we can add a rule to the bottom of the grammar:

5 - 2 * 8 / (3 + 4 * 2)

0: S -> E

1: E -> E + T
2: E -> E - T
3: E -> T

4: T -> T * F
5: T -> T / F
6: T -> F

7: F -> P ^ F
8: F -> P

9: P -> ~ P
10: P -> M

11: M -> 0 | 1 | ... | 9 |
12: M -> (E)